



Geert Van de Putte
Dean Keister

WebSphere Message Broker V6, Best Practices Guide: Bullet Proofing Message Flows

If you do not know what you are trying to accomplish, the task is even harder to do. If you do not know how to do the task, it is even more difficult. How many times have you started to develop a solution before understanding the requirements — input data formats, the arrival rates, processing needs, and the expected output? How much do you really know about the tools that you are using and the full power that those tools can bring to the solution? To make the best choices when designing and implementing message flows, you need a clear understanding of the requirements and the power of the tools that you are using.

Message flows should be *bullet proof*, meaning that the design should provide the required functions and should also prevent errors from disrupting normal operations. For example, a message flow is bullet proof when:

- ▶ A problem is recognized early and its impact is minimized.
- ▶ A problem is diagnosed on the fly and corrective action taken.
- ▶ Information about what was happening at the point of failure is collected and recorded.
- ▶ The message flow takes proactive steps to notify someone about the problem with details about the problem.

This paper examines some of the capabilities of WebSphere® Message Broker and shows how to include these capabilities in your message flows. We use variations on a sample message flow using an MQInput node, several Compute nodes, and MQOutput nodes to show error path options, how the broker handles errors in various nodes, and the information available for analysis. We do not discuss all possibilities and options in this paper. Many of the discussion points rely heavily on, and have borrowed information that is provided in, the WebSphere Message Broker documentation and help files. When you need more detail or when questions arise, refer to this material.

Error handling principles

The following are the main principles used to develop bullet proof message flows:

1. Create a backup queue and set a backout threshold on the input queue.

In many cases, a backup queue can help you know where to look for a failed message.

2. Place most of your error handling in the Catch flow of the MQInput node to contain most of the general purpose error recording and notification.

Using the Catch flow of the MQInput node puts common error handling in one place and leaves Failure flows on Compute nodes and Catch flows on TryCatch nodes so that you can provide specific actions in response to individual situations.

3. Put a Throw node at the end of the Catch flow.

You should put a Throw node at the end of the Catch flow to backout any uncommitted work and to leave a record in the Event log.

4. Let someone know.

Remember, when you provide a Catch flow or Failure flow, you are assuming total responsibility. You should always let someone else know what you are doing in regards to error handling.

Your flow design can be simple or quite complex. The options for the MQInput node are extensive because this node must deal with persistent and non-persistent messages and with transactional and non-transactional flows. The outcome can also be impacted by configuration options for WebSphere MQ. Because the choice is yours, there are no fixed rules. However, there are design options and other factors to consider when designing each message flow.

Design considerations

The following are a number of design considerations that you can apply when designing a message flow:

- ▶ Connect the Failure terminal of any node to a sequence of nodes that processes the node's internal exception (the Failure flow).
- ▶ Connect the Catch terminal of the input node or a TryCatch node to a sequence of nodes that processes exceptions that are generated beyond it (the Catch flow).
- ▶ Insert one or more TryCatch nodes at specific points in the message flow to catch and process exceptions that are generated by the flow that is connected to the Try terminal.
- ▶ Include a Throw node or code an ESQL THROW statement to generate an exception.
- ▶ Ensure that all messages that are received by an MQInput node are processed within a transaction or are not processed within a transaction.
- ▶ Ensure that all messages that are received by an MQInput node are persistent or are not.

Understand the flow sequence

When an exception is detected *within a node*, the message and the exception information are propagated to the node's Failure terminal. If the node does not have a Failure terminal or if it is not connected, the broker throws an exception and returns control to the closest previous node that can process the exception. This node can be a TryCatch node or the MQInput node.

A message is propagated to a Catch terminal only if it has first been propagated beyond the node (for example, to the nodes that are connected to the Out terminal).

If an MQInput node detects an internal error, its behavior is slightly different. If the Failure terminal is not connected, it attempts to put the message to the input queue's backout requeue queue or (if that is not specified) to the dead letter queue of the broker's queue manager. If a message cannot be put to the specified queue, a loop can occur.

If you do not connect the Catch terminal of the MQInput node, you can connect the Failure terminal and provide a Failure flow to handle exceptions that are generated by the node. The Failure flow is invoked immediately when an exception occurs in the MQInput node.

The MQInput Failure flow is also invoked if an exception is generated beyond the MQInput node in either Out flow (when the Catch terminal is not connected) or a Catch flow, if the flow is transactional or the message is persistent and if the reinstatement of the message on the input queue causes the backout count to reach the backout threshold.

Other considerations to keep in mind

Other considerations for your message flow include the following:

- ▶ If you connect a Failure or Catch terminal in any node, you are indicating that the flow handles all exceptions that are generated and passed to the flow. The broker performs no rollback and takes no action unless there is an exception within that flow. If you want any rollback action after an exception has been raised and caught, you must provide this in the flow.
- ▶ When a message is propagated to the Failure or Catch terminal, the node creates and populates a new ExceptionList with information about the error. The ExceptionList is propagated as part of the message tree.
- ▶ The MQInput treats transactional and non-transactional messages differently.
- ▶ If you include a Trace node that specifies \$Root or \$Body, the complete message is parsed. This might generate parser errors that are not otherwise detected.
- ▶ If a node propagates a message to a Catch flow and another exception occurs that returns control to the same node again, the node handles the message as though the Catch terminal is not connected.
- ▶ If you do not connect either Failure or Catch terminals of the MQInput node, the broker provides default processing.
- ▶ If you have a common procedure for handling particular errors, you might find it useful to create a subflow that includes the sequence of nodes that is required. Then, you can include this subflow when you need that action to be taken.

What happens when exceptions occur

The rules controlling the paths taken and what information is available when errors occur is complex and can be confusing. What happens and what information is available depends on where the error occurs, what nodes are used, which terminals are wired, whether the flow is transactional or the message persistent, and if there are multiple errors. Each of these factors can affect the results.

To see how these error-handling principles work with real message flows, we use the sample message flow shown in Figure 1 and see what happens with a single error when different nodes are present with different Failure and Catch terminal connections. We also look at what happens when additional errors occur in Catch or Failure flows.

Possible error paths

Figure 1 shows the message flow that is used to demonstrate various error paths. The specific path traversed is based on where errors occur, what terminals are connected, the transactional state of the message flow, and the persistence of the message.

The notation in Figure 1 allows for a number of different message flows and indicates the following:

- ▶ The dashed lines between terminals indicate that in some examples there is a connection between terminals and in others there is no connection.
- ▶ The nodes that have the red *x* icon indicate points where errors might occur in different examples.
- ▶ The dashed box indicates that the TryCatch node might or might not be present in a particular example. (This notation becomes more clear when we look at the cases in detail.)

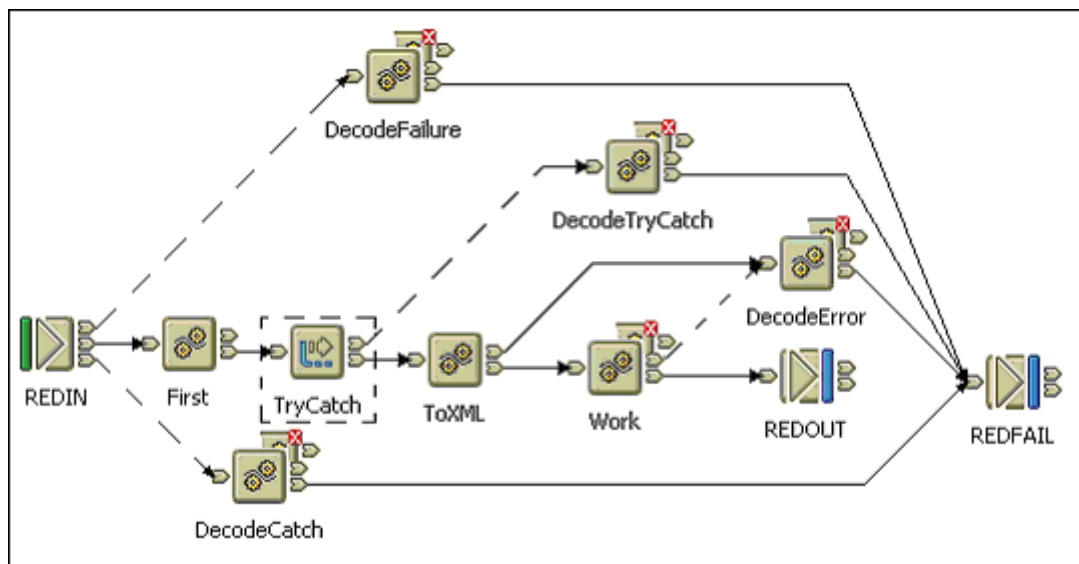


Figure 1 Sample message flow

If there are no errors in the flow, an MRM message placed on the REDIN queue is transformed to XML and the results are placed on REDOUT. It does not matter if any Failure or Catch terminals are connected because there are no errors.

To examine what happens to the Environment and LocalEnvironment trees, each Compute node modifies the Environment and LocalEnvironment variables. For example, the Compute node First creates a LocalEnvironment variable called UserData.Path and an Environment variable called UserData.Path. Each subsequent node adds its name to the path so that a trace of these variables at the end of a successful execution shows the path that the message flow has taken (Environment.Variable.UserData.Path = 'First-ToXML-Work').

Failure outcomes

By examining the possible failure scenarios, you find there are 41 possible error paths. The good news is that the number of different outcomes for the paths is a more manageable nine. Each path has a specific outcome for the Environment tree, the LocalEnvironment tree, the ExceptionList tree, the Event log, and the Flow Results. When designing a failure flow for Failure or Catch flows, you must understand the path that got you there so that you know what information is available. The flowchart in Figure 2 shows the conditions that lead to these different outcomes.

The Environment and LocalEnvironment trees might contain information, an ExceptionList tree might be attached, the Event log might be updated, there might be a message placed on the backout queue, there might be a loop, and a variety of Failure or Catch flows might be executed. The outcome all depends on which terminals are connected, where the error or errors occur, whether there is a TryCatch node, the transactional nature of the flow or the persistence state of the message, and the application logic within the Failure flow.

The flowchart in Figure 2 allows you to locate a specific case based on a number of design and configuration options. When an error occurs, the specific case is determined by your flow design. When you design a flow and errors occur, a specific path applies, and there is a unique outcome. Table 1 on page 7 describes each outcome.

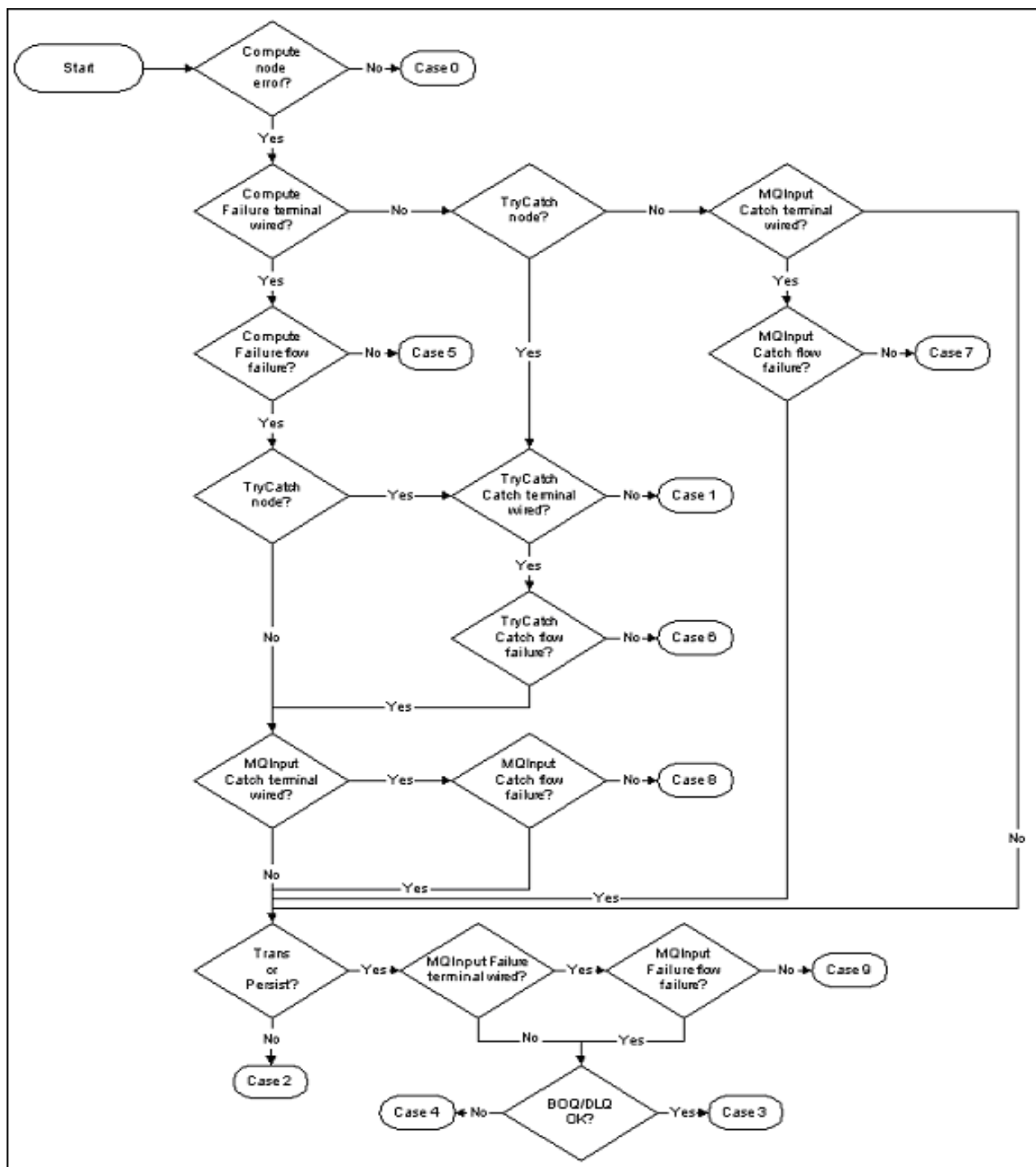


Figure 2 Failure paths flowchart

Examination of outcomes

Table 1 shows the state of the Environment, LocalEnvironment, and ExceptionList trees. In addition, the Flow Results column shows the result of the broker execution. There is also an indication of what is in the Event Log. Although the Event Log is not accessible, knowing what is provided might influence what additional information can be useful for the error handling flow to provide.

Table 1 Outcomes for various message flows

	Environment Variable Changes	Local Environment Variable Changes	ExceptionList Tree	Flow Results	Event Log
Case 0 - No errors	N/A	N/A	N/A	Message put to REDOUT	Nothing in Event Log
Case 1 - No Failure or Catch terminal wired with TryCatch node in flow	N/A	N/A	N/A	Message discarded	Nothing in Event Log
Case 2 - Flow is non-transactional and message is non-persistent, no Failure or Catch terminals wired and no TryCatch node in flow	N/A	N/A	N/A	Message discarded	All errors encountered in Event Log
Case 3 - Flow is transactional or message is persistent, no Failure or Catch terminals wired, and no TryCatch node in flow	N/A	N/A	N/A	Message put on Backout Queue (BOQ) / Dead Letter Queue (DLQ)	All errors encountered and indication of message move to BOQ/DLQ in Event Log
Case 4 - Problem putting a message to the BOQ / DLQ	N/A	N/A	N/A	Retry Loop	All errors encountered and failure of message move to BOQ/DLQ in Event Log
Case 5 - Compute node Failure flow handles error	As modified	As modified before <i>Error</i> node	Describes Error	<i>Error</i> node Failure flow	Nothing in Event Log
Case 6 - TryCatch Catch flow handles error	As modified	As modified before TryCatch node	Describes Error	TryCatch Catch flow	Nothing in Event Log
Case 7 - MQInput Catch flow handles error	As modified	Discarded	Describes Error	MQInput Catch flow	Nothing in Event Log
Case 8 - MQInput Catch flow handles last error	As modified	Discarded	Describes last Error	MQInput Catch flow	Compute error and detection of additional error in Event Log
Case 9 - MQInput Failure flow handles last error	Discarded	Discarded	Indicates failed message dequeued	MQInput Failure flow driven	All errors encountered in Event Log

Case 1 and Case 2 show what happens when the Work node in Figure 1 on page 4 fails but with no Failure or Catch terminals connected:

- ▶ Case 1 - No Failure or Catch terminals wired with TryCatch node in flow. The message is discarded, even if the message is persistent or the flow transactional and *nothing is placed in the Event log*! It is assumed that the message flow handles the error itself.
- ▶ Case 2 - Flow is non-transactional and message is non-persistent, no Failure or Catch terminals wired, and no TryCatch node in flow. The message is discarded and entries are placed in the Event log.

Case 3 and Case 4 highlight backout processing. Backout queue processing only comes into play when the flow is transactional or the message is persistent. If a backout queue (BOQ) is specified, backout rules apply to that queue and the dead-letter queue (DLQ) is never considered. So the cases studied here only reference a backout queue where appropriate.

- ▶ Case 3 - Flow is transactional or message is persistent, no Failure or Catch terminals wired, and no TryCatch node in flow. The Event log is updated and the message is placed on the backout queue.
- ▶ Case 4 - A message would normally be put on the backout queue but there is a problem. The Event Log is updated, but the message cannot be discarded. So, it is continually reinstated on the input queue. The flow stays in this loop until either the cause of the original failure situation clears, allowing the message to be processed, or until the backout queue finally accepts the message, at which time Case 3 processing takes place.

Under normal circumstances, a failure is passed to either a Failure or Catch flow. Case 5 through Case 7 cover the situations where these flows successfully process the original error:

- ▶ Case 5 - Compute node Failure flow handles error. Nothing is placed in the Event Log, and the Failure flow is responsible for all error recovery. This responsibility includes committing or backing out any updates, putting entries in the Event Log (via Trace node), notifications, and so on.

A review of Case 5 in Table 1 shows that the Environment variable contains all changes made up to the point of the failure. So, any changes made prior to the error within the node Work are available to the node DecodeError. However, the LocalEnvironment variable only contains changes made prior to the node Work. So, any changes made within the node Work are lost. In addition, an ExceptionList tree is added to the message tree and can be examined within the Failure flow.

- ▶ Case 6 - TryCatch Catch flow handles error. This case is similar to Case 5. Nothing is placed in the Event Log and the Catch flow is responsible for all error recovery.
- ▶ Case 7 - The MQInput Catch flow handles the error. This case is similar to Case 5. Nothing is placed in the Event log and the Catch flow is responsible for all error recovery.

There might be additional errors in Failure or Catch flows after the original error, which are covered in Case 8 and Case 9, as well as Case 2 through Case 4.

- ▶ Case 8 - The MQInput Catch flow handles the last error. The Event Log contains information about the prior error, while the ExceptionList contains information about the last error. The most probable situation is when there is an error in the Failure flow of the Work node with no TryCatch node, but the MQInput node Catch terminal is wired and its flow is successful. Other paths leading to Case 8 are shown in Figure 2 on page 6.
- ▶ Case 9 - The MQInput Failure flow handles the last error. The Event Log contains all errors. If the MQInput Failure flow is successful, you have Case 9. If the MQInput Failure flow fails, backout processing applies (Case 3 and Case 4). Other paths leading to Case 9 are shown in Figure 2 on page 6.

First failure data capture

Anticipating possible points of error allows for identifying and capturing data that would aid in analysis should an error occur. With this analysis, you can then put in place a plan for keeping track of this data. A useful place to put this information is in the Environment tree, `Environment.Variables.UserData`, as shown in Figure 4 on page 13. Planning for how this data might be used — either by an application or a person — can help the designer provide better failure data.

Application data

When errors occur, there might be information that can help clarify what the error is and what the message flow was doing at the time of the error. By examining Table 1 on page 7, you can see that when a Compute node Failure flow or a Catch flow gets control, the Environment tree contains all modifications made prior to that failure. With careful design, information relevant to flow execution can be placed into this tree throughout the message flow so that the information can be available to a Failure flow or Catch flow. Useful information might include execution paths (as shown in the Sample Message Flow described earlier), input from database queries, results of calculations, records processed, and so on. What data should be captured is dependent upon the application. However, the goal is to try to capture the state of the application at the point of failure.

The ExceptionList tree

In addition to data that is provided by the application, the broker might also provide useful information. Table 1 on page 7 indicates that the ExceptionList tree is populated with information about the last error when a Compute node Failure flow or a Catch flow gets control. Code might be provided to attempt recovery for certain types of errors. Example 1 shows how ESQL code loops through the ExceptionList tree and keeps the error Label, Number, and Text in the Environment tree.

Example 1 Processing the ExceptionList

```
declare abc REFERENCE TO InputExceptionList.*[1];
WHILE abc.Number IS NOT NULL do
set Environment.Variables.BrokerData.LastError.Label = abc.Label;
set Environment.Variables.BrokerData.LastError.Number =
cast(abc.Number as char);
    set Environment.Variables.BrokerData.LastError.Text = abc.Text;
    set Path = Path || '.*[<]';
    -- Move start to the last child of the field to which it
    -- currently points
    MOVE abc LASTCHILD;
END WHILE;
```

A trace of the Environment tree after adding user data and extracting information from the ExceptionList tree might show output similar to that shown in Example 2.

Example 2 Trace output of the Environment tree

```
(0x01000000):Variables = (
  (0x01000000):UserData = (
    (0x03000000):Trail = 'First-ToXML-CreateError-Decode Error'
    (0x03000000):RecCount = '1'
  )
  (0x01000000):BrokerData = (
    (0x03000000):Label = 'OneRow.Simple.Work'
    (0x03000000):Number = '2450'
    (0x03000000):Text = 'Divide by zero calculating '%1 / %2''
```

)
)

Error recovery considerations

Armed with knowledge of error paths and the information that is available for examination, it is time to see what actions you can use when handling errors. Your application might detect errors in the data content that should be noted and isolated but that might not end the flow. In this situation, you might want to continue processing. In other situations, you might want to perform recovery or terminate the flow gracefully. You must decide when you design the message flow which recovery action is appropriate. However, you should also consider recording the information about the error condition and performing some error notification to make it easier for someone to analyze the problem and to take appropriate action.

Attempt recovery

Your application might try to recover from certain errors (for example, an inconsistency of some type in the input data). The Compute node that detects this error might record information in the Environment tree (for example, `Environment.Variables.BrokerData.Number`) and throw an error. Then, in a Failure flow or Catch flow, you might have an error handling error flow to extract the error number for the last error in the ExceptionList tree and route the message to different error handling procedures based on the error number. In your error handling procedure, you could then examine the Environment tree and take appropriate action.

Message commit or backout

As a rule of thumb, issue a Throw node as the last node of your Failure flow or Catch flow to backout any uncommitted messages on a queue. When a message is put on a queue, it is not available until committed. This commit occurs immediately if the message flow is non-transactional, the output queue has its Transactional attribute set to No, or the Failure flow or Catch flow ends without a Throw node. In other cases, the message is backed out, and the message is discarded.

Clean up after failure

If a message flow is transactional and you are handling errors in a Failure flow or Catch flow, you take full responsibility for the transaction. This responsibility might include committing database updates or committing messages on queues. If you want to backout all uncommitted database updates or remove uncommitted messages from a queue, you must include a Throw node as the last node of your Failure flow or Catch flow. If you want to commit all updates, then end the flow without throwing an error.

Error notification considerations

Providing notification of an error and documenting its occurrence is a very important aspect of the message flow process as well. At times, it can be very difficult to re-create the error situation. However, by providing detailed information about the error when it happens, the error might be solved easier and quicker.

Trace to the Event Log

Probably the simplest action to take regarding documentation of the error is to augment the Event log with information pertinent to the application. If the application is recording

information in the Environment tree, this information might be useful to help in problem determination. Examples of this type of trace might include the list of nodes that executed prior to the failure or state information pertinent to the application. Figure 3 shows an example of Trace node properties for sending information to the Event log and the Event log entry. The choice of data is determined by the designer of the flow.

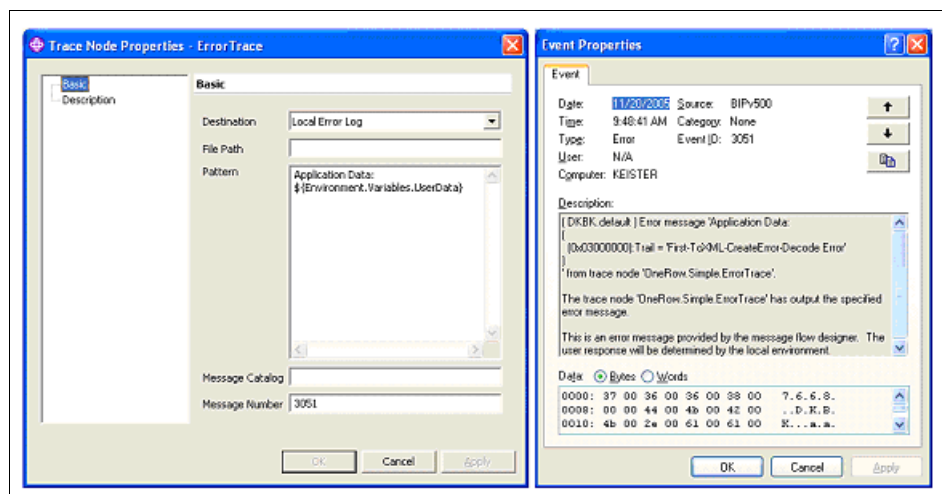


Figure 3 Trace to Event Log

Throw an exception

When you use the Throw statement or include a Throw node, an entry is placed in the Event log. If you use the Throw statement, you can provide application data as part of the Event log parameters. However, if the Failure terminal of the Compute node throwing the error is connected and the Failure path ends successfully, no rollback actions are invoked because the processing in the Failure path is part of the same unit of work. Adding a Throw node or a Throw statement at the end of the Catch flow avoids this behavior because it forces a rollback. If you use a Throw node, backout actions are performed, but you cannot include data from within the flow in the Event log. Using Trace and Throw gives you a variety of ways to control the actions and information needed.

Put a message on a queue

When designing your Failure or Catch flow, you might want to put a message to a queue as part of your error handling. The presence of this message might enable a program that monitors the queue to perform some notification. If the message flow is transactional or the message persistent, a message is put on the backout queue if defined and a Throw node is encountered. However, there might be cases when no message is put on the backout queue or this might not be all that is needed, for example:

- ▶ There is no backout or dead-letter queue defined.
- ▶ A message flow is transactional or the message persistent and no errors are thrown.
- ▶ A message flow is non-transactional and the message non-persistent.
- ▶ In some production environments, only certain users might be allowed access to the broker queue manager and queues. So, any messages placed on the backout queue might not be available to those who need to perform problem determination.

In these situations, it might be useful to put a message that contains diagnostic information to a queue that is available to those who can use it for problem determination. This queue might need to be a remote queue, but it allows the responsible group more control for notification

and faster access to information. How to use Trace and Throw nodes and how to put a message to a queue is explained in “MQInput node catch flow design” on page 12.

Pro-active notification

You should strongly consider proactive notification of failures. Recording error information is useful, but action can be taken only when someone becomes aware of the failure. There are two common options and a third, more sophisticated, option:

- ▶ **Third-party products**

Third-party products provide standard mechanisms for detection of messages in queues (for example, in the dead-letter queue) with many built-in actions such as paging or sending e-mail messages. However, these tools are often oriented to production systems and the organizations that support them. It might be difficult for the group that is responsible for the message flows to tailor the action of the third-party tool for their specific needs or have these tools available in development and test systems.

- ▶ **Send an e-mail**

As a companion to third-party products, or as an alternative, consider using the e-mail SupportPac™. It can usually be installed and configured to work with minimal outside assistance. The SupportPac can be used in development, test, and production environments and provides detailed First Failure Data for problem determination, in many cases eliminating the need to re-create errors. With experience gained during development, the data captured can be evaluated and optimized to provide the best possible information when promoted to the production environment. See “Using Sendmail” on page 15.

- ▶ **Start a Workflow**

WebSphere MQ Workflow processes can be started by putting an XML message onto a queue. With planning, a workflow process can be started to initiate recovery and analysis. This workflow process can be used in combination with the other notification techniques and is a useful technique for meeting service level commitments.

Performance considerations

Performance requirements can directly influence the design of your message flow. In situations where performance is of prime importance, avoid Trace nodes or saving failure information in the main path. Instead, record information in the Environment tree and leave the rest to a Catch or Failure flow. If performance is critical, put minimum logic in the Catch or Failure flow. As an alternative for providing more robust failure processing within the flow, consider collecting information, putting it into a message, and putting that message on a queue that is processed by a separate message flow. This process frees up the production flow to handle more messages.

Putting it into practice

The examples that we discuss here are based on real-world experiences. We have, however, sanitized them to protect any proprietary information and have simplified them to make the bullet proofing points clear.

MQInput node catch flow design

The Catch flow shown in Figure 4 can be as general or as specific as needed. The designer of this message flow placed some useful information in the Environment tree, Environment.Variables.UserData, where the children of UserData contain information that

was deemed to be of value in error analysis, such as the nodes that were executed and state variables.

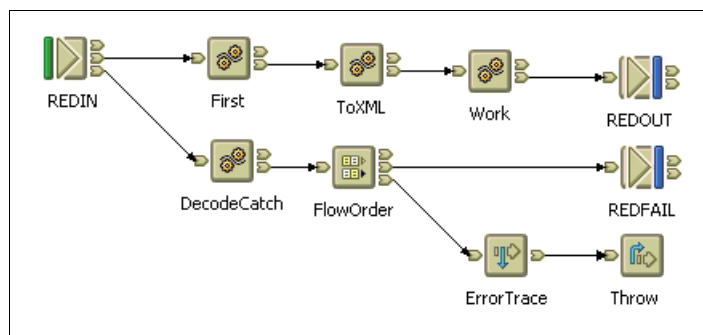


Figure 4 MQInput node catch flow

The Decode Catch ESQL decodes the ExceptionList tree and builds an XML message containing user data and the ExceptionList tree. First, this message is put to an output queue. Then, a Trace node puts an entry in the Event log followed by a Throw node to backout any database changes and uncommitted messages.

One important point to remember is that if an error is thrown, any messages put on queues by flows that are transactional are discarded when the Throw node is processed. To ensure that this does not happen, set the Transaction mode to No as shown in Figure 5.

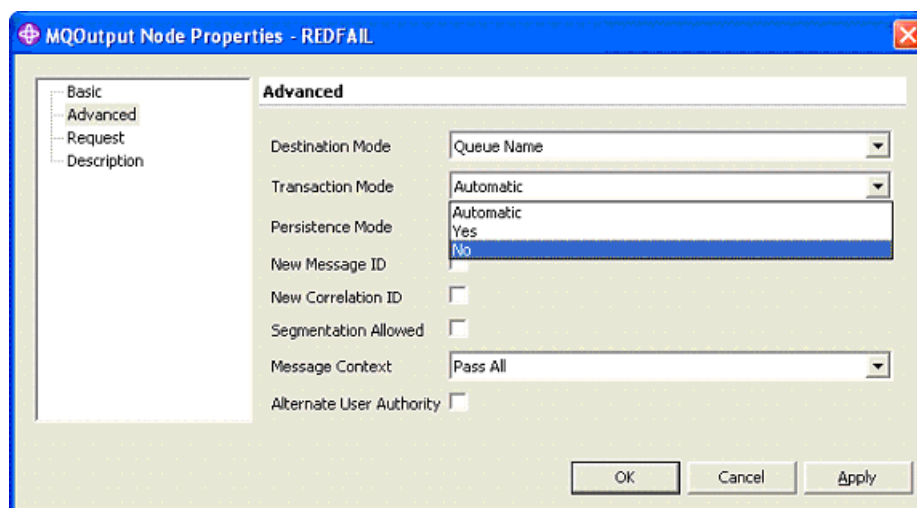


Figure 5 Set Transaction Mode in the MQOutput node

The message placed on REDFAIL contains information that the flow designer deemed valuable for problem diagnosis. This queue might be remote, there might be a third-party tool monitoring that queue, or there might be a message flow that picks up the message to perform additional error recovery and notification, such as sending an e-mail. Using this technique off loads the recovery actions to a separate flow and leaves the primary flow available to process additional messages immediately.

Example 3 shows the ESQL from the DecodeCatch node decoding of the ExceptionList tree and placing it in Environment.Variables.BrokerData. In addition, it updates the application's trail of nodes held in Environment.Variables.UserData.Trail. Finally, it builds an output message and builds a body that contains UserData and BrokerData, which are placed on the REDFAIL queue.

Example 3 Decode the ExceptionList and build error message

```
-- Loop through the exception list children
declare abc REFERENCE TO InputExceptionList.*[1];
set Path = 'Environment.Variables.BrokerData';
WHILE abc.Number IS NOT NULL do
    -- keep track of the deepest error
    set Environment.Variables.BrokerData.LastError.Label = abc.Label;
    set Environment.Variables.BrokerData.LastError.Number = cast(abc.Number as char);
    set Environment.Variables.BrokerData.LastError.Text = abc.Text;
    -- record the error information in tree format
    EVAL( 'set ' || Path || '.' || FIELDNAME(abc) || '.Label = ''' || abc.Label ||
''';');
    --Error text might contain quotes so must replace one single quote (') with
    -- two single quotes (') so EVAL interprets properly
    set work = abc.Text;
    set work = replace(work, '''', ''''');
    EVAL( 'set ' || Path || '.' || FIELDNAME(abc) || '.Text = ''' || work || ''';');
    EVAL( 'set ' || Path || '.' || FIELDNAME(abc) || '.Number = ''' || cast(abc.Number
as char) || ''';');
    set insertNo = insertBase;
    WHILE FIELDNAME(abc.*[insertNo]) = 'Insert' do
        EVAL('set ' || Path || '.' || FIELDNAME(abc) || '.Insert[' ||
        cast(insertNo - insertBase + 1 as character) || '].Text = abc.*[' ||
        cast(insertNo as char) || '].*[2];');
        set insertNo = insertNo + 1;
    END WHILE;
    set Path = Path || '.*[<]';
    -- Move start to the last child of the field to which it currently points
    MOVE abc LASTCHILD;
END WHILE;
set Environment.Variables.UserData.Trail =
coalesce(Environment.Variables.UserData.Trail, ' ') || '-Decode Error';
set Path = 'Environment.Variables.BrokerData';
CALL CopyMessageHeaders();
set OutputRoot.XML.Message.Environment.Variables.UserData = Environment.Variables.UserData;
set OutputRoot.XML.Message.Environment.Variables.BrokerData =
Environment.Variables.BrokerData;
```

Example 4 shows the body of the message built by the ESQL code shown in Example 3.

Example 4 Generated XML message

```
<Message>
  <Environment>
    <Variables>
      <UserData>
        <Trail>First-ToXML-CreateError-Decode Error</Trail>
      </UserData>
      <BrokerData>
        <LastError>
          <Label>OneRow.Simple.Work</Label>
          <Number>2450</Number>
          <Text>Divide by zero calculating &apos;%1 / %2&apos;</Text>
        </LastError>
        <RecoverableException>
          <Label>OneRow.Simple.Work</Label>
          <Text>Caught exception and rethrowing</Text>
          <Number>2230</Number>
        </RecoverableException>
        <Label>OneRow.Simple.Work</Label>
      </BrokerData>
    </Variables>
  </Environment>
</Message>
```

```

    <Text>Error detected, rethrowing</Text>
    <Number>2488</Number>
    <Insert>
      <Text>OneRow.CreateError.Main</Text>
    </Insert>
    <Insert>
      <Text>8.3</Text>
    </Insert>
    <Insert>
      <Text>SET k = 1 / CAST(OutputRoot.XML.Hi.One AS INTEGER);</Text>
    </Insert>
    <RecoverableException>
      <Label>OneRow.Simple.Work</Label>
      <Text>error evaluating expression</Text>
      <Number>2439</Number>
      <Insert>
        <Text>OneRow.CreateError.Main</Text>
      </Insert>
      <Insert>
        <Text>8.12</Text>
      </Insert>
      <Insert>
        <Text>1 / CAST(OutputRoot.XML.Hi.One AS INTEGER)</Text>
      </Insert>
      <RecoverableException>
        <Label>OneRow.Simple.Work</Label>
        <Text>Divide by zero calculating &apos;%1 / %2&apos;</Text>
        <Number>2450</Number>
        <Insert>
          <Text>1 / 0</Text>
        </Insert>
      </RecoverableException>
    </RecoverableException>
  </RecoverableException>
</BrokerData>
</Variables>
</Environment>
<LocalEnvironment/>
</Message>

```

This message contains significant information about the exact cause of the error — in this case, a Divide by Zero error was encountered in the node Work using values 1/0. The ExceptionList tree along with well thought out user data might be enough to solve the problem or at least provide significant clues without the need to re-create the error.

Using Sendmail

Much of the information shown in the message created by the MQInput Catch Flow example could appear in the Event log. However, someone must be alerted to the failure. Using the Sendmail SupportPac to send someone an e-mail is a simple and inexpensive solution. You can download this SupportPac at no charge from:

http://www-1.ibm.com/support/docview.wss?rs=203&uid=swg24000600&loc=en_US&cs=utf-8&lang=en

The notification is immediate and the data is at hand. Someone can be working on the problem before most are even aware there was a failure. If the performance of this flow is critical, you might consider building a separate message flow for sending e-mails and passing a request message to it from this flow, as shown in Figure 6.

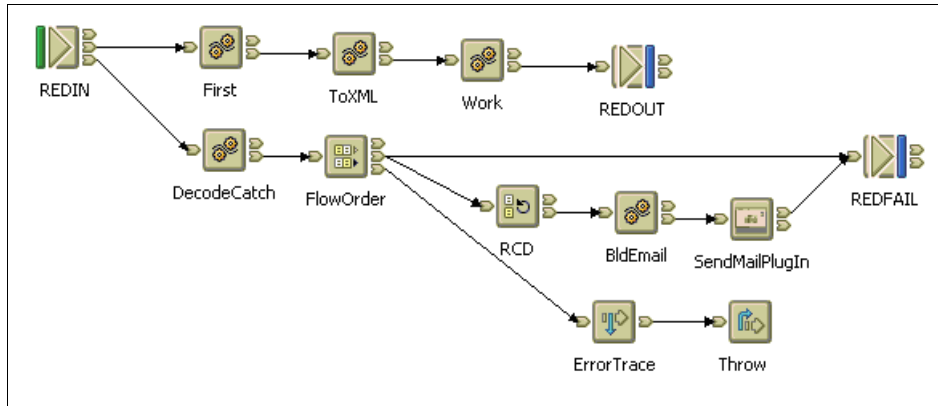


Figure 6 Using Sendmail in a message flow

The output message from DecodeCatch is in XML format. To be able to add this message to the text of an e-mail, we first have to convert it to an MRM message with one field. Then, the BldEmail node takes this field containing the XML message as a string and builds the e-mail request. The Sendmail plug-in requires lines of the message to be no more than 1024 characters so the ESQL must enforce this line limit. An e-mail might look similar to that shown in Figure 7.

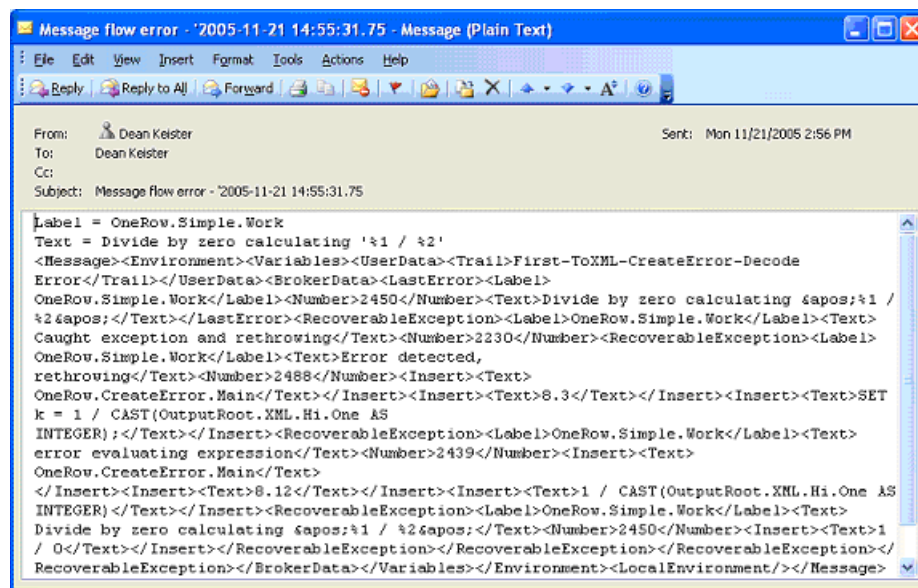


Figure 7 Message flow generated e-mail

The e-mail can contain anything that you feel is appropriate. In this instance, the subject contains a time stamp, and the body contains both summary information for the label that detected the error and the text that shows the type of error. In addition, the full XML message as shown in Figure 4 on page 13 is included. Example 5 shows the ESQL that was used to build the Sendmail message.

Example 5 ESQL to prepare an e-mail message

```

CALL CopyMessageHeaders();
set OutputRoot.XML.Message.To = 'dkeister@centerprise.com';
set OutputRoot.XML.Message.From = 'dkeister@centerprise.com';
set OutputRoot.XML.Message.Subject =
substring(cast(CURRENT_TIMESTAMP as char) from 11 for 23) ||
  
```



```

' - Error in ' || coalesce(Environment.Variables.BrokerData.LastError.Label, 'unknown')
||
' - ' || coalesce(Environment.Variables.BrokerData.LastError.Text, 'unknown');
set msgbody = r.LFROW.ROW;
set i = position('</' in msgbody);
set j = position('>' in msgbody from i);
while i > 0 do
  set j = position('>' in msgbody from i);
  if j < 1024 then
    set m = j;-- remember position of '>' for </ tag
    set l = l + 1;-- index for checking next '</'
  else
    -- this '</' tag will go over the 1024 limit for sendmail
    -- line length so put from the last remembered position
    set OutputRoot.XML.Message.Body.Line[k] = left(msgbody, m);
    set k = k + 1;
    set msgbody = substring(msgbody from m + 1); -- clear out
    set l = 1; -- reset
  end if;
  -- find next instance of closing tag. Eval required because of repeat
  EVAL('set i = position('</' in msgbody from i REPEAT ' || cast(l as char) || ')');
end while;
set OutputRoot.XML.Message.Body.Line[k] = msgbody;

```

With this technique, when a failure is encountered, the message flow provides pertinent information and pro-actively notifies someone about the problem.

Database recovery

When a message flow with database access first starts, a handle to the database is established and cached. When a message arrives on the MQInput node this handle is used, and the overhead of establishing a connection is avoided. If the database is shut down and restarted or if the connection handle times out, the next time the flow receives a message, it tries to use the handle that is no longer valid and a database error occurs. Other errors are permanent and there is no recovery. In those situations when a new database handle is required, you can stop or restart the message flow or re-drive the node accessing the database. Figure 8 shows a representative flow.

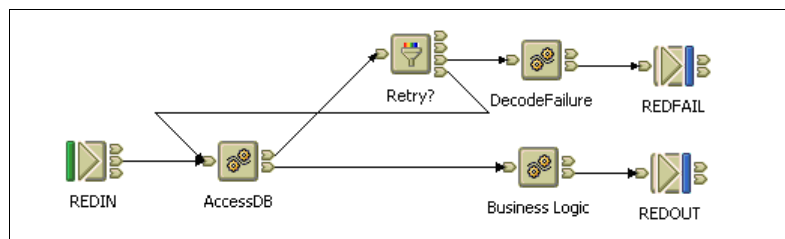


Figure 8 Database recovery

To handle database errors in the AccessDB node, you must access the node's properties and deselect **Throw exception on database error**.

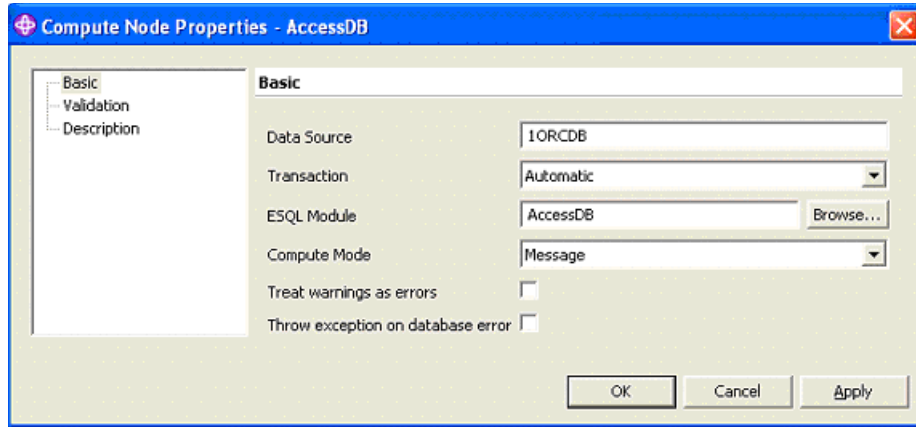


Figure 9 Handling database exceptions

This allows you to now handle database errors. In the ESQL shown in Example 6, any non-zero SQLCODE indicates an error. The information is gathered and an error is thrown to drive the Failure flow.

Example 6 ESQL to handle database errors

```

/*
Action: Throw error 2962 for any DB error to drive failure terminal
  I have found the following errors recoverable
  SQL 3113 = EOF on communication channel (among many)
  SQL 3114 = not connected to ORACLE
  SQL 12571 = TNS packet writer failure
*/
set Environment.Variables.UserData.DataSelect[] = (
  SELECT  F.FEEDDEFID,
          F.HdrCount,
          F.BrokerHost,
          F.Work
  FROM    Database.MQIADMIN.FLOWCONTROL as F
  WHERE   F.FEEDDEFID = 'DEFAULT' or
          F.FEEDDEFID = Environment.Variables.UserData.FeedDefId);
if SQLCODE <> 0 then
  set Environment.Variables.UserData.ErrCode = '2962';
  set Environment.Variables.UserData.SQLSTATE = cast(SQLSTATE as char);
  set Environment.Variables.UserData.SQLCODE = cast(SQLCODE as char);
  set Environment.Variables.UserData.SQLERRORTEXT = SQLERRORTEXT;
  set Environment.Variables.UserData.SQLNATIVEERROR = cast(SQLNATIVEERROR as char);
  throw USER EXCEPTION MESSAGE 2962 VALUES
    (Environment.Variables.UserData.SQLSTATE,
     Environment.Variables.UserData.SQLCODE,
     Environment.Variables.UserData.SQLERRORTEXT,
     Environment.Variables.UserData.SQLNATIVEERROR,
     'FLOWCONTROL table access failure');
end if;
-- Put business logic here if database access is successful

```

If the Failure terminal simply connects to the In terminal, a loop occurs until the database error clears. If the database error is permanent, the flow loops continuously. To prevent this continuous looping, you must add some logic in the Failure flow. The Filter node checks the error and performs loop control.

Example 7 ESQL for Filter node

```
-- If this is a 'caught' DB error then check retry processing
if Environment.Variables.UserData.ErrCode = '2962' then
-- Retry processing sets a retry count on the first entry and
-- drives the true terminal on each subsequent execution, the
-- retry count is decremented.
    -- If the count does not go to zero, the true terminal is driven.
    -- When the count goes to zero, the false terminal is driven.
    if Environment.Variables.UserData.DBRetryCount is null then
-- if this is first retry,
-- Environment.Variables.UserData.DBRetryCount will be null
        set Environment.Variables.UserData.DBRetryCount = 3;
        return true;
    else
-- if this is not the first retry, DBRetryCount will
-- contain a count.
        if Environment.Variables.UserData.DBRetryCount < 1 then
            -- if the retry count is exhausted, stop retry
            return false;
        else
            -- decrement the DBRetryCount
            set Environment.Variables.UserData.DBRetryCount =
Environment.Variables.UserData.DBRetryCount - 1;
            return true;
        end if;
    end if;
else
    return false;
end if;
RETURN TRUE;
```

If the error is not a database error, processing goes immediately to DecodeFailure. If this is the first thrown database error, the Environment.Variables.UserData.DBRetryCount should be null. It is set to the retry count 3 and control is passed back to the In terminal of the AccessDB node. If the database access fails again, this time the DBRetry count variable is decremented. This repeats until the connection is made successfully, in which case AccessDB continues normal processing.

If the DBRetry count decrements to zero (0), the False terminal flow gets control and DecodeFailure processing takes place. Notice that the variables that we use in this scenario are saved in Environment.Variables.UserData. The database error conditions and retry counts that are used in the Failure flow might be passed via an e-mail or other notification technique if the failure is permanent. If the flow does recover from a Database error, logic could be inserted in the Out flow of AccessDB to check fields in the UserData area and an information e-mail sent that a database error occurred but that recovery was successful.

Conclusion

Anticipating and planning for errors can have near-term and long-term benefits. Error handling is often an afterthought in many development cycles. However, it can become critical quickly when errors occur in production environments. Solutions can be simple or sophisticated, but you should consider your solutions right from the beginning. Bullet-proof message flows will not only help solve development problems, they will help fine-tune for production. A word of caution — it is easy to go overboard and over-design the solution. Ensure that the discussion for error handling is a part of your normal development process.

Appendix: Retry processing

When the number of retries has reached the backout threshold limit, the message flow attempts to propagate the message through the Failure terminal, if that is connected. If you have not connected the Failure terminal, the node attempts to put the message to either the backout or dead letter queue, if it is defined.

If the backout threshold has not been reached, the node gets the message from the queue again. If this fails, this is handled as an internal error (described in “MQInput node catch flow design” on page 12). If it succeeds, the node propagates the message to the Out flow.

If the backout threshold has been reached:

1. If you have connected the Failure terminal, the node propagates the message to that terminal. You must handle the error in the Failure flow.
2. If you have not connected the Failure terminal, the node attempts to put the message on an available queue, in order of preference:
 - a. The message is put on the input queue's backout requeue name (queue attribute BOQNAME), if one is defined.
 - b. If the backout queue is not defined, or it cannot be identified by the node, the message is put on the dead letter queue (DLQ), if one is defined.
 - c. If the message cannot be put on either of these queues because there is an MQPUT error (including queue does not exist), or because they cannot be identified by the node, it cannot be handled safely without risk of loss.

The message cannot be discarded. Therefore the message flow continues to attempt to backout the message. It records the error situation by writing errors to the local error log. A second indication of this error is the continual incrementing of the BackoutCount of the message in the input queue.

If this situation has occurred because neither queue exists, you can define one of the backout queues mentioned above. If the condition preventing the message from being processed has cleared, you can temporarily increase the value of the backout threshold (BOTHRESH) attribute. This forces the message through normal processing.

The team that wrote this Redpaper

This Redpaper was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Poughkeepsie Center.

Geert Van de Putte is a Consulting IT Specialist at the International Technical Support Organization, Raleigh Center. He is a subject matter expert for messaging and business integration and has nine years of experience in the design and implementation of WebSphere Business Integration solutions. He has published several redbooks about messaging and business integration. Geert has also taught several classes about these subjects. Before joining the ITSO, Geert worked at IBM® Global Services, Belgium, where he designed and implemented EAI solutions for customers in many industries. Geert holds a Master of Information Technology degree from the University of Ghent in Belgium.

Dean Keister Dean Keister is Director, Business Integration at Centerprise Services, Inc. with responsibility for design and development of systems that enable customers to integrate business activities combining both new and existing applications in real time and to quickly automate business processes across the enterprise. Before joining Centerprise, Dean held technical and managerial positions in IBM with responsibilities for developing and deploying computer networks, transaction processing systems, and messaging and queuing middleware. He is an IBM Certified Solutions Expert for WebSphere MQ, IBM WebSphere Message Broker, and WebSphere MQ Workflow.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Send us your comments in one of the following ways:

- ▶ Use the online **Contact us** review redbook form found at:
ibm.com/redbooks
- ▶ Send your comments in an e-mail to:
redbook@us.ibm.com

© Copyright International Business Machines Corporation 2006. All rights reserved.


Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

- Mail your comments to:
IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400 U.S.A.



Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

@server®
@server®
Redbooks (logo) ™

IBM®
Redbooks™
SupportPac™

WebSphere®

Other company, product, or service names may be trademarks or service marks of others.